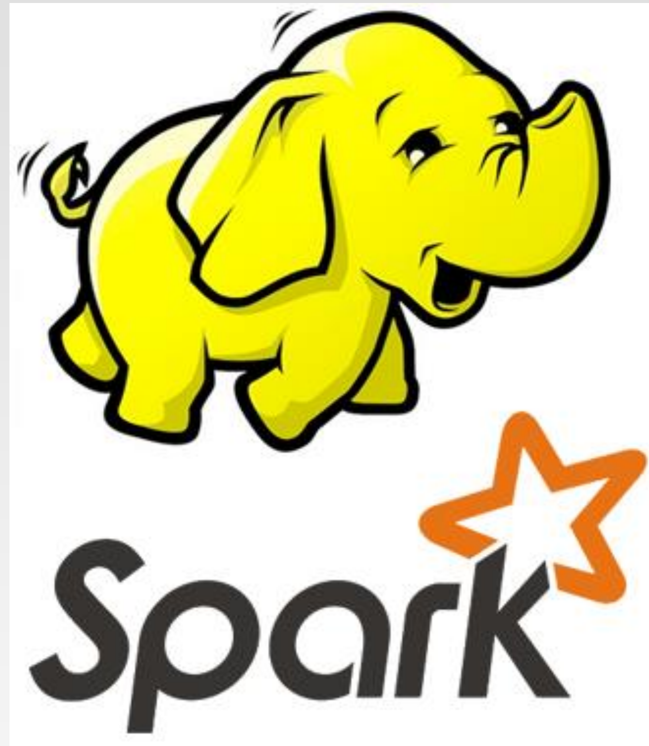
# COMP9313: Big Data Management




Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

# Final exam

- Final written exam (50 pts)
- Six questions in total on five topics
- Four hours (Do not wait for the last minute to submit!)
- Online exam. Submit through Moodle
- If you are ill on the day of the exam, do not attend the exam – will not accept any medical special consideration claims from people who already attempted the exam.

# myExperience Survey

## Give us a grade

UNSW has a new student course survey – **myExperience**

Look out for your email invitation and for links in Moodle

Fill out the survey to help us improve your courses and teaching at UNSW

# Chapters Required in Exam

- Hadoop MapReduce (Chapters 1, 2, and 3)
  - HDFS
  - MapReduce Concepts and Mechanism
  - MapReduce algorithm design
- Spark (Chapters 4 and 5)
  - RDD
  - DataFrame
- Mining Data Streams (Chapter 6)
- Finding Similar Items (Chapter 7)
- Graph Data Management (Chapter 8)

# Exam Questions

- Question 1: HDFS, MapReduce, and Spark concepts
- Question 2: MapReduce algorithm design (pseudo-code only)
- Question 3: Spark algorithm design
  - RDD
  - DataFrame
- Question 4 Finding Similar Items
  - Shingling, Min Hashing, LSH
- Question 5 Mining Data Streams
  - Sampling, DGIM, Bloom filter, Finding frequent items ~~FM-sketch~~
- Question 6 Graph Data Management

# Question 0

- (a) (2 marks) Explain the data flow in MapReduce using the word count problem as an example.

- (b) (2 marks) Explain the data flow in Spark using the word count problem as an example.

# Map and Reduce Functions

- Programmers specify two functions:
  - **map** $(k_1, v_1) \rightarrow$ list $[<k_2, v_2>]$
    - Map transforms the input into key-value pairs to process
  - **reduce** $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - Reduce aggregates the list of values for each key
    - All values with the same key are sent to the same reducer
- Optionally, also:
  - combine $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  - partition ($k_2$, number of partitions) $\rightarrow$ partition for $k_2$
    - Often a simple hash of the key, e.g., hash($k_2$) mod n
    - Divides up key space for parallel reduce operations
  - Grouping comparator: controls which keys are grouped together for a single call to Reducer.reduce() function
- The execution framework handles everything else…

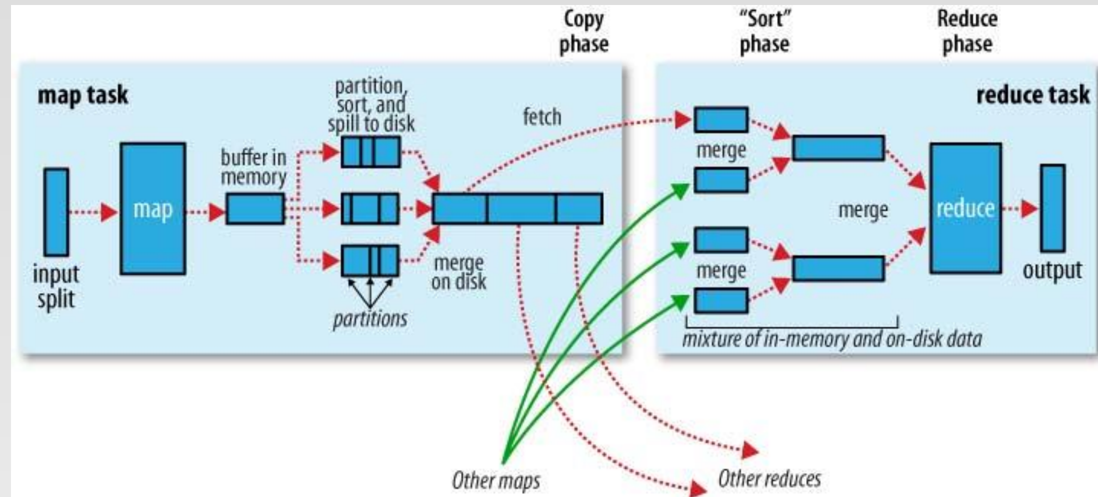
# Combiners

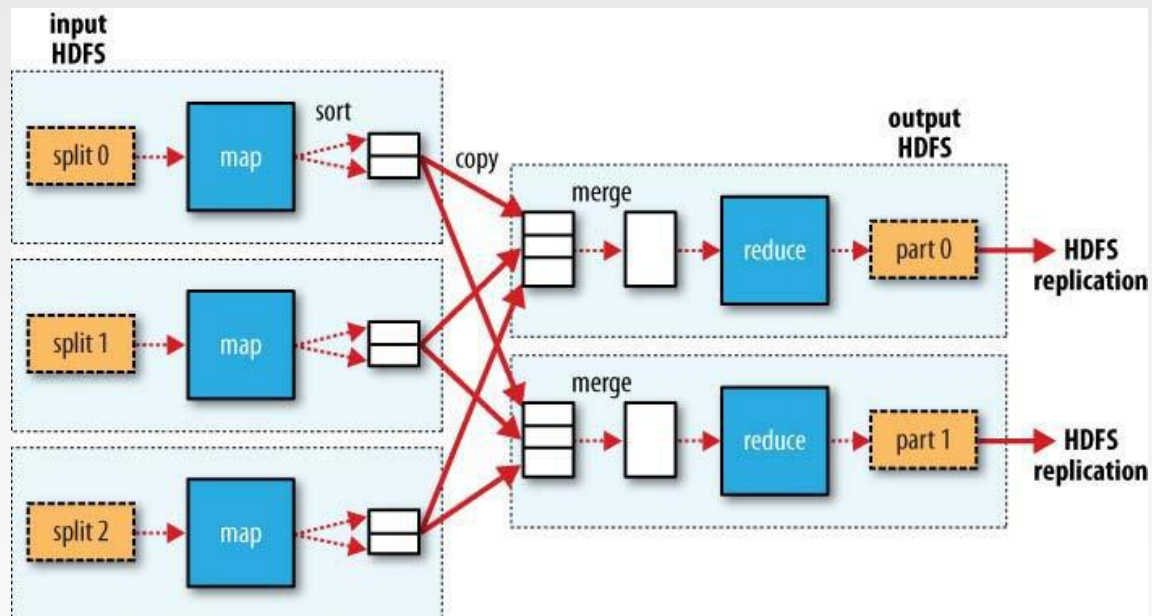
- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key *k*
  - E.g., popular words in the word count example
- Combiners are a general mechanism to reduce the amount of intermediate data, thus saving network time
  - They could be thought of as "mini-reducers"
- Warning!
  - The use of combiners must be thought carefully
    - Optional in Hadoop: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output
  - Works only if reduce function is commutative and associative
    - In general, reducer and combiner are not interchangeable

# Partitioner

- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
    - This controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.
- System uses HashPartitioner by default:
  - hash(key) mod R
- Sometimes useful to override the hash function:
  - E.g., ***hash(hostname(URL)) mod R*** ensures URLs from a host end up in the same output file
- Job sets Partitioner implementation (in Main)

# MapReduce Data Flow



# MapReduce Algorithm Design Patterns

- In-mapper combining, where the functionality of the combiner is moved into the mapper.
  - Scalability issue (not suitable for huge data) : More memory required for a mapper to store intermediate results
- The related patterns “pairs” and “stripes” for keeping track of joint events from a large number of observations.
- “Order inversion”, where the main idea is to convert the sequencing of computations into a sorting problem.
  - You need to guarantee that all key-value pairs relevant to the same term are sent to the same reducer
- “Value-to-key conversion”, which provides a scalable solution for secondary sort.

# Sample Questions

- Assume that you are given a data set crawled from a location-based social network, in which each line of the data is in format of (userID, a list of locations the user has visited <loc1, loc2, …>). Your task is to compute for each location the set of users who have visited it, and the users are sorted in ascending order according to their IDs.

# Solution

```
class Question1
        method map(self, userID, list of locations)
                foreach loc in the list of locations
                        Emit(“loc, userID”, “”)

        method reduce_init(self)
                current_loc = “”
                current_list = []

        method reduce(self, key, value)
                loc, userID = key.split(“,”)
                if loc != current_loc
                     if current_loc!=“”
                          Emit(current_loc, current_list)
                     current_list = []
                     current_list.add(userID)
                     current_loc=loc
                else
                     current_list.add(userID)

        method reduce_final(self)
                Emit(current_loc, current_list)

In JOBCONF, configure:
        'mapreduce.map.output.key.field.separator':’,’,
        'mapreduce.partition.keypartitioner.options':'-k1,1’,
        'mapreduce.partition.keycomparator.options':'-k1,1 -k2,2'
```

# Sample Questions

- Given a table shown as below, find out the person(s) with the maximum salary in each department (employees could have the same salary).

| EmployeeID | Name | DepartmentID | Salary |
|---|---|---|---|
| 001 | Emma | 1 | 100,000 |
| 002 | Helen | 2 | 85,000 |
| 003 | Jack | 3 | 85,000 |
| 004 | James | 1 | 110,000 |

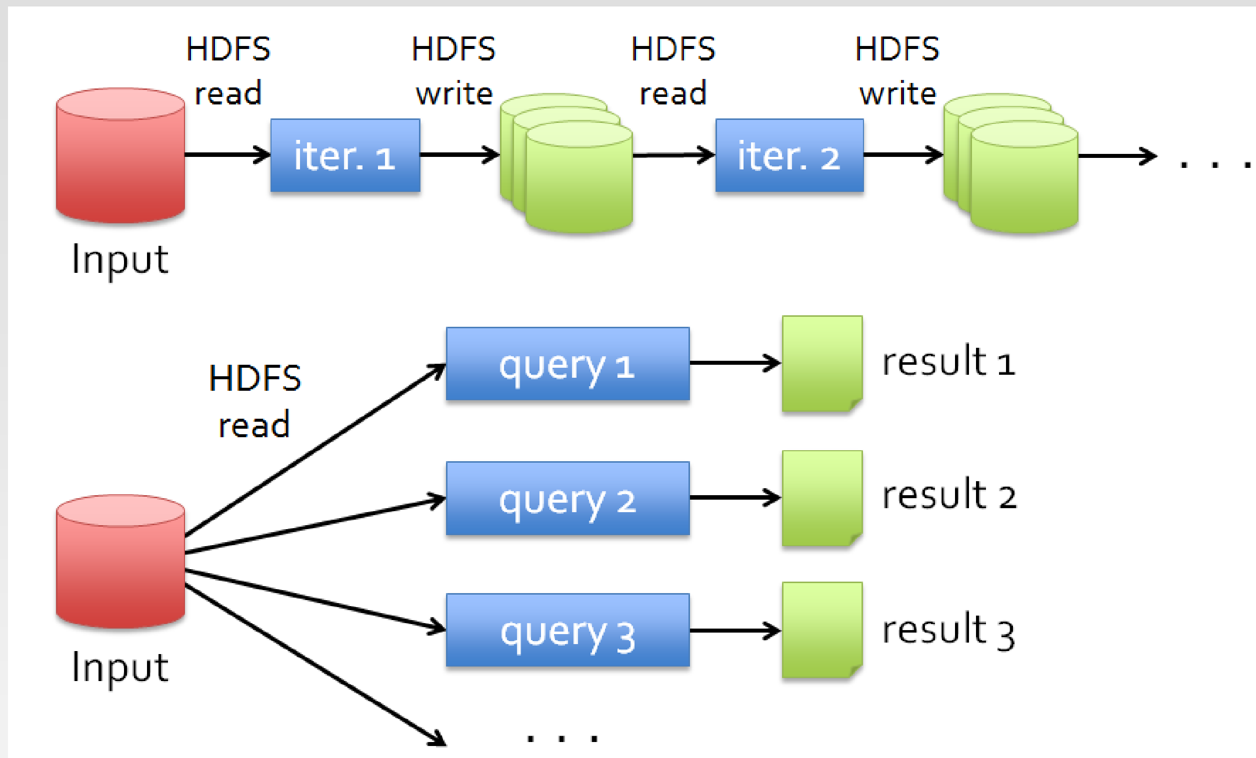
- Solution:
  - Mapper: for each record, Emit(department + "," + salary, name)
  - Combiner: find out all persons with the local maximum salary for each department
  - Reducer: receives data ordered by (department, salary), the first one is the maximum salary in a department. Check the next one until reaching a smaller salary and ignore all remaining. Save all persons with this maximum salary in the department
  - JOBCONF: key partitioned by "-k1,1", sorted by "-k1,1 -k2,2n"

# Sample Questions

- Given a large text dataset, find the top-k frequent terms (considering that you can utilize multiple reducers, and the efficiency of your method is evaluated).

- Solution:
  - Two rounds:
    - First round compute term frequency in multiple reducers, and each reducer only stores local top-k.
    - Second round get the local top-k and compute the final top-k using a single reducer.

# Data Sharing in MapReduce

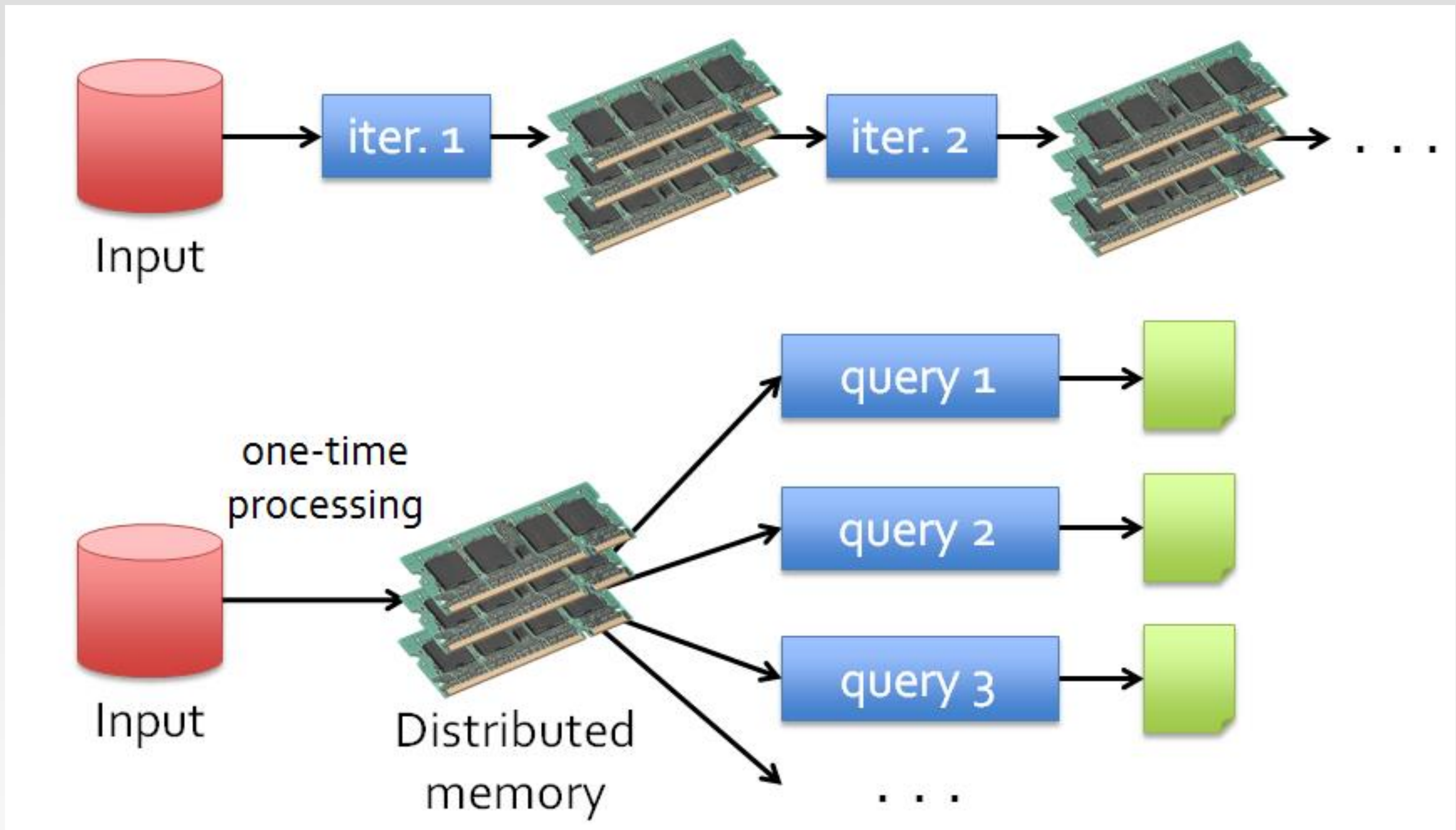



**Slow** due to replication, serialization, and disk IO

- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

# Data Sharing in Spark Using RDD

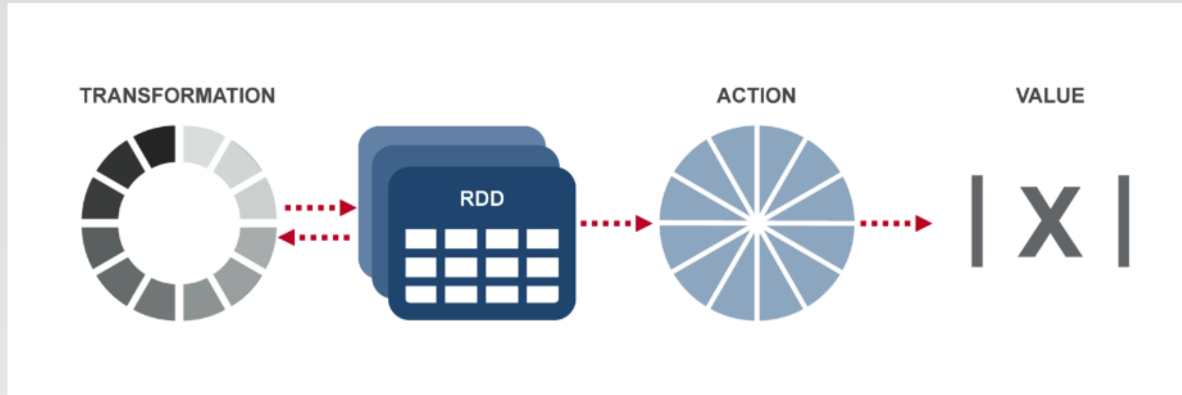



**10-100×** faster than network and disk

# What is RDD

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12
  - RDD is a **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner.
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**
  - Data residing on multiple nodes in a cluster.
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel.

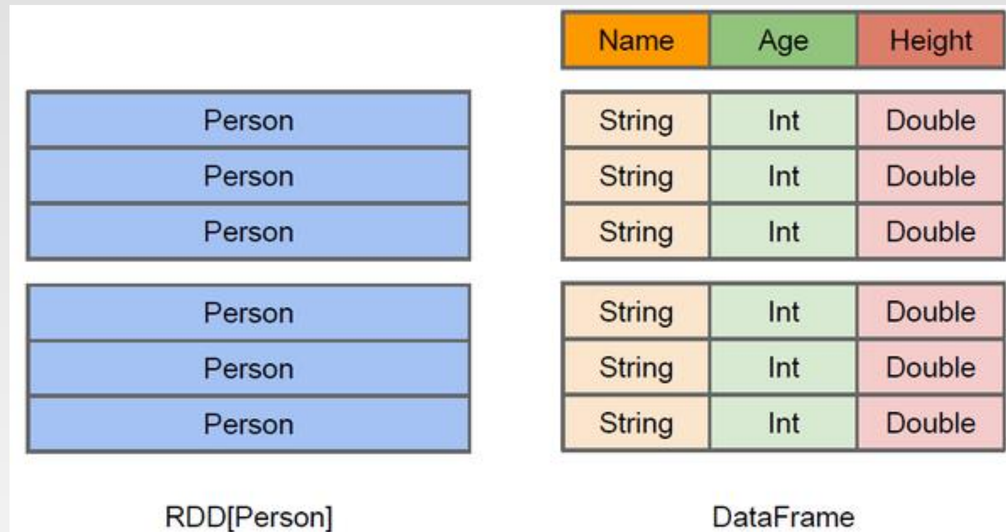
# RDD Operations




- **Transformation:** returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, filter, join, etc.*
- **Action:** evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

# DataFrame

- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema




  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus, Spark SQL query optimizer can target optimization

# Sample Questions

- **RDD:** Given a large text file, your task is to find out the top-k most frequent co-occurring term pairs. The co-occurrence of (w, u) is defined as: u and w appear in the same line (this also means that (w, u) and (u, w) are treated equally). Your Spark program should generate a list of ***k*** key-value pairs ranked in descending order according to the frequencies, where the keys are the pair of terms and the values are the co-occurring frequencies (**Hint:** you need to define a function which takes an array of terms as input and generate all possible pairs).

```
val textFile = sc.textFile(inputFile)
val words = textFile.map(_.split(" ").toLowerCase)

// fill your code here, and store the result in a pair RDD topk

topk.foreach(x => println(x._1, x._2))
```

# Sample Questions

- Given a set of marks from different courses (the input format is as shown in the left column), the task is to: compute average marks for every course and sort the result by course_name in alphabetical order.

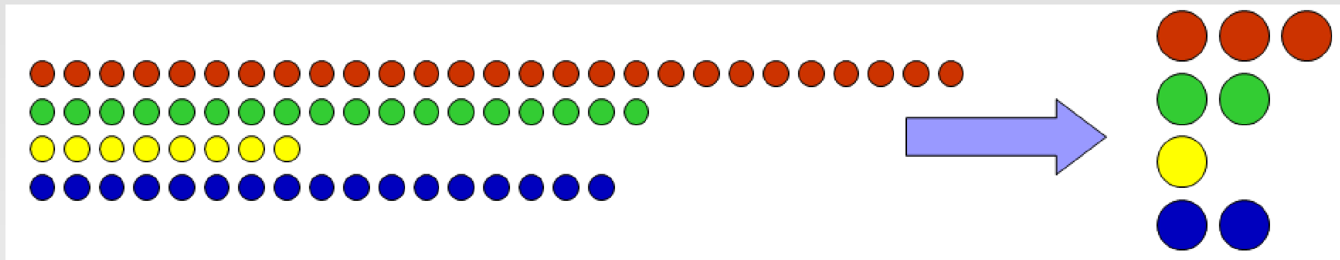
| Input: | Output: |
| --- | --- |
| student1:course1,90;course2,92;course3,80;course4,79;course5,93<br>student2:course1,92;course2,77;course5,85<br>student3:course3,64;course4,97;course5,82 | course1:91<br>course2:84.5<br>course3:72<br>course4:88<br>course5:86.67 |

- Solution:

```
fileDF = spark.read.text("file:///home/comp9313/tinydoc")

student = fileDF.select(split(fileDF['value'], ':').getItem(0).alias('sid'), split(fileDF['value'],
':').getItem(1).alias('courses’))

scDF = student.withColumn('course', explode(split('courses', ';’)))

scDF2 = scDF.select(split(scDF['course'], ',').getItem(0).alias('cname'), split(scDF['course'],
',').getItem(1).alias('mark’))

avgDF = scDF2.groupBy('cname').agg(avg('mark')).orderBy('cname')
```

# Sampling Data Streams

- Since we can not store the entire stream, one obvious approach is to store a **sample**



- Two different problems:
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10)
    - As the stream grows the sample also gets bigger
  - **(2)** Maintain a **random sample of fixed size** over a potentially infinite stream
    - As the stream grows, the sample is of fixed size
    - At any "time" ***t*** we would like a random sample of ***s*** elements
      - **What is the property of the sample we want to maintain?** For all time steps ***t***, each of ***t*** elements seen so far has equal probability of being sampled

# Sample Questions

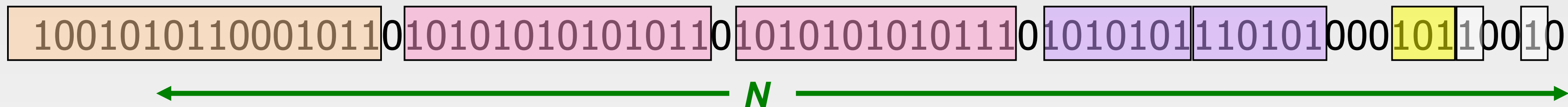
- Use an example to explain the reservoir sampling algorithm

  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the $n^{th}$ element arrives (***n*** **>** ***s***)
    - With probability ***s/n***, keep the $n^{th}$ element, else discard it
    - If we picked the $n^{th}$ element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random

# DGIM Algorithm

- **Idea:** Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

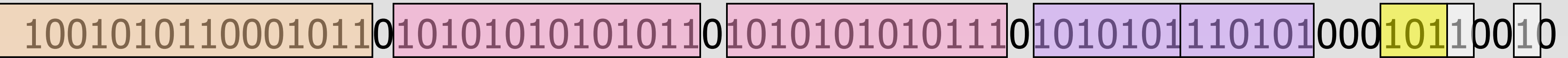
- When there are few 1s in the window, block sizes stay small, so errors are small




- Timestamps:
  - Each bit in the stream has a timestamp, starting from **1**, **2,** …
  - Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $O(\log_2 N)$ bits
    - E.g., given the windows size 40 (***N***), timestamp 123 will be recorded as 3, and thus the encoding is on 3 rather than 123

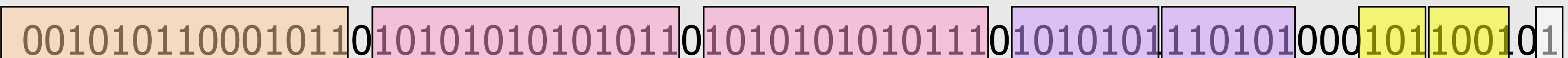
# Example: Updating Buckets

**Current state of the stream:**

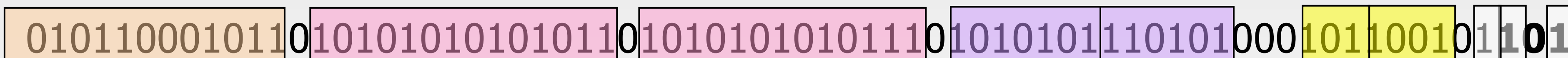



**Bit of value 1 arrives**

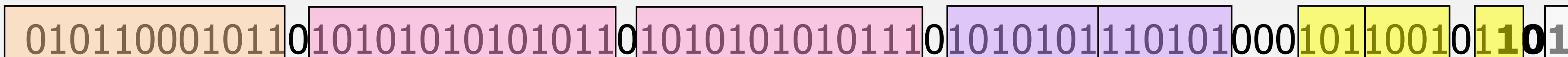
**Two white buckets get merged into a yellow bucket**



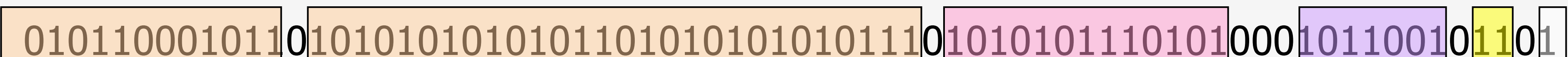

**Next bit 1 arrives, new orange white is created, then 0 comes, then 1:**




**Buckets get merged…**




**State of the buckets after merging**



# Sample Questions

Suppose we are maintaining a count of 1s using the DGIM method. We represent a bucket by (i, t), where i is the number of 1s in the bucket and t is the bucket timestamp (time of the most recent 1).

Consider that the current time is 200, window size is 60, and the current list of buckets is: (16, 148) (8, 162) (8, 177) (4, 183) (2, 192) (1, 197) (1, 200). At the next ten clocks, 201 through 210, the stream has 0101010101. What will the sequence of buckets be at the end of these ten inputs?

# Solution

- There are 5 1s in the stream. Each one will update to windows to be:
  - (1) (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(1, 197)(1, 200), (1, 202)

    => (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), (1, 202)
  - (2) (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), (1, 202), (1, 204)
  - (3) (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), (1, 202), (1, 204), (1; 206)

    => (16, 148)(8, 162)(8, 177)(4, 183)(2, 192)(2, 200), (2, 204), (1, 206)

    => (16, 148)(8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206)
  - (4) Windows Size is 60, so (16,148) should be dropped.

    (16, 148)(8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206), (1, 208) => (8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206), (1, 208)
  - (5) (8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (1, 206), (1, 208), (1, 210)

    => (8, 162)(8, 177)(4, 183)(4, 200), (2, 204), (2, 208), (1, 210)

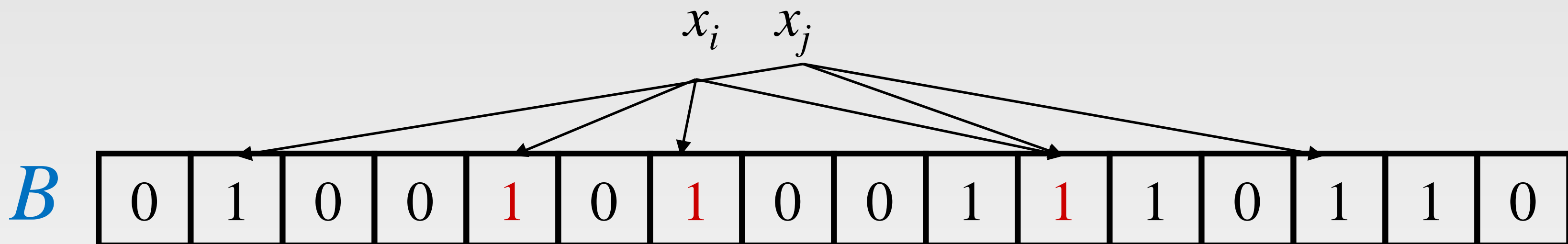
# Bloom Filter

- Consider: **|S| = *m*, |B| = *n***
- Use $k$ independent hash functions $h_1, \ldots, h_k$
- **Initialization:**
  - Set **B** to all **0s**
  - Hash each element $s \in S$ using each hash function $h_i$, set $B[h_i(s)] = 1$ (for each $i = 1, \ldots, k$)
- **Run-time:**
  - When a stream element with key $x$ arrives
    - If $B[h_i(x)] = 1$ for all $i = 1, \ldots, k$ then declare that $x$ is in $S$
      - That is, $x$ hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element $x$

# Counting Bloom Filter

- Bloom filters can handle insertions, but not deletions.

- If deleting $x_i$ means resetting 1s to 0s, then deleting $x_i$ will "delete" $x_j$.




- Can Bloom filters handle deletions?
  - Use Counting Bloom Filters to track insertions/deletions

# Counting Bloom Filters

Start with an $n$ bit array, filled with 0s.

| $B$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Hash each item $x_j$ in $S$ for $k$ times. If $H_i(x_j) = a$, add 1 to $B[a]$.

| $B$ | 0 | 3 | 0 | 0 | 1 | 0 | 2 | 0 | 0 | 3 | 2 | 1 | 0 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

To delete $x_j$ decrement the corresponding counters.

| $B$ | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 3 | 2 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Can obtain a corresponding Bloom filter by reducing to 0/1.

| $B$ | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Sample Questions

- Consider a Bloom filter of size m = 7 (i.e., 7 bits) and 2 hash functions that both take a string (lowercase) as input:

  $h1(str) = \sum_{(c \text{ in } str)}(c-'a') \bmod 7$

  $h2(str) = str.length \bmod 7$

Here, c - ‘a’ is used to compute the position of the letter c in the 26 alphabetical letters, e.g., h1(“bd”) = (1 + 3) mod 7 = 4.

  - (i) Given a set of string S = {“hi”, “big”, “data”}, show the update of the Bloom filter
  - (ii) Given a string “spark”, use the Bloom filter to check whether it is contained in S.
  - (iii) Given S in (i) and the Bloom filter with 7 bits, what is the percentage of the false positive probability (a correct expression is sufficient: you need not give the actual number)?

# Solution

- (i)

| | hi | big | data |
|---|---|---|---|
| **h1** | **(7+8) mod 7 = 1** | **(1+8+6) mod 7 = 1** | **(3+0+19+0) mod 7 = 1** |
| **h2** | **2 mod 7 = 2** | **3 mod 7 = 3** | **4 mod 7 = 4** |

- (ii) h1 (spark) = (18 + 15 + 0 + 17 + 10) mod 7 = 4

  h2 (spark) = 5 mod 7 = 5

  Not in S since the 4th bit is 1 but the 5th bit is 0

- (iii) k – # of hash functions; m – # of inserting elements; n - # of bits

$$\left(1 - e^{-\frac{km}{n}}\right)^k = 0.3313$$

# Approximate Heavy Hitters

- A more general problem: find all elements with counts > n/k (k>=2)
  - There can be at most k-1 such values; and there might be none
  - Trivial if we have enough storage

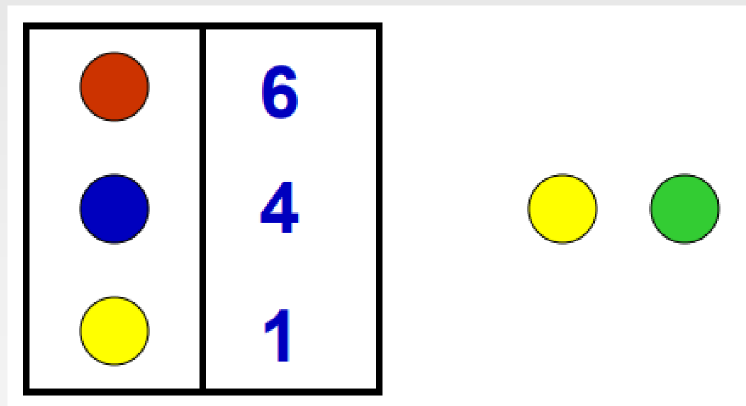
- There is no exact algorithm that solves the Heavy Hitters problems in one pass while using a sublinear amount of auxiliary space

- Relaxation, the ε-approximate heavy hitters problem:
  - If an element has count > n/k, it must be reported, together with its estimated count with (absolute) error < εn
  - If an element has count < (1/k − ε) n, it cannot be reported
  - For elements in between, don't care

- In fact, we will estimate all counts with at most εn error

# Misra-Gries Algorithm

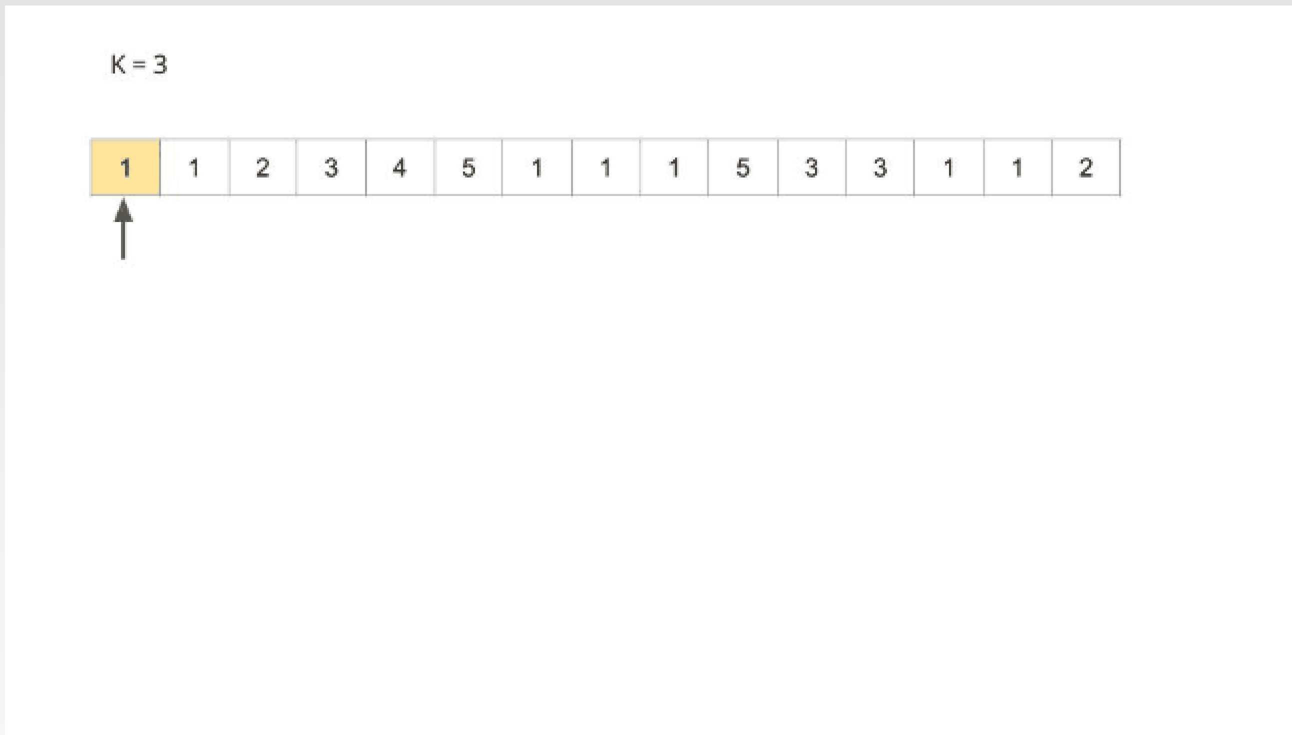
- Keep k-1 different candidates in hand (thus with space O(k))
- For each element in stream:
  - If item is monitored, increase its counter
  - Else, if < k-1 items monitored, add new element with count 1
  - Else, decrease all counts by 1, and delete element with count 0




- Each decrease can be charged against k arrivals of different items, so no item with frequency N/k is missed
- But false positive (elements with count smaller than n/k) may appear in the result

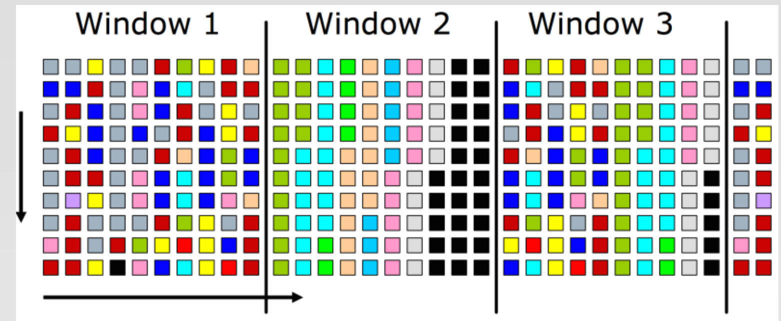
# Misra-Gries Algorithm

- [1,1,2,3,4,5,1,1,1,5,3,3,1,1,2] with k=3, we want to find element that occurred more than 15/3 = 5 times.

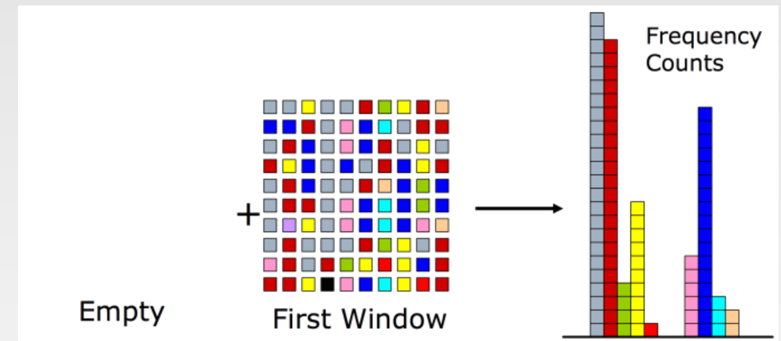


# Lossy Counting

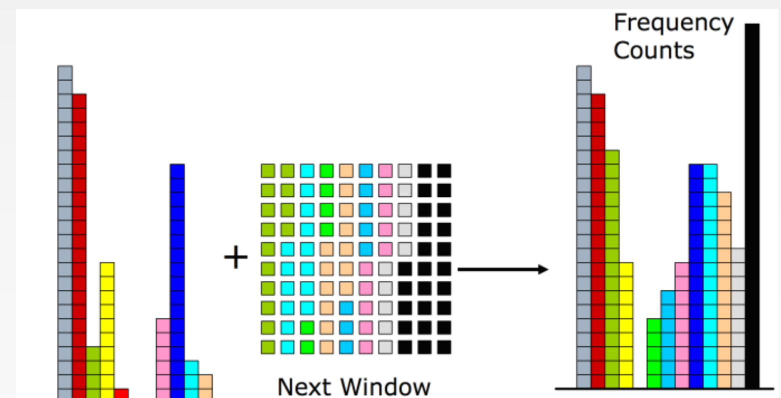
- Step 1: Divide the incoming data stream into windows, and each window contains 1/ε elements



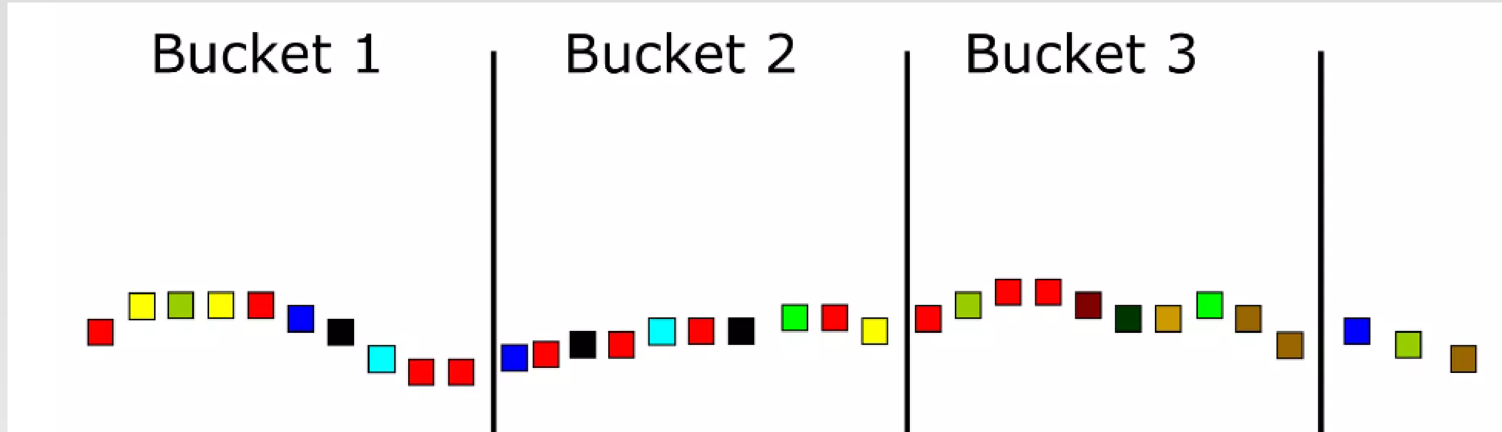

- Step 2: Increment the frequency count of each item according to the new window values. After each window, decrement all counters by 1. Drop elements with counter 0.

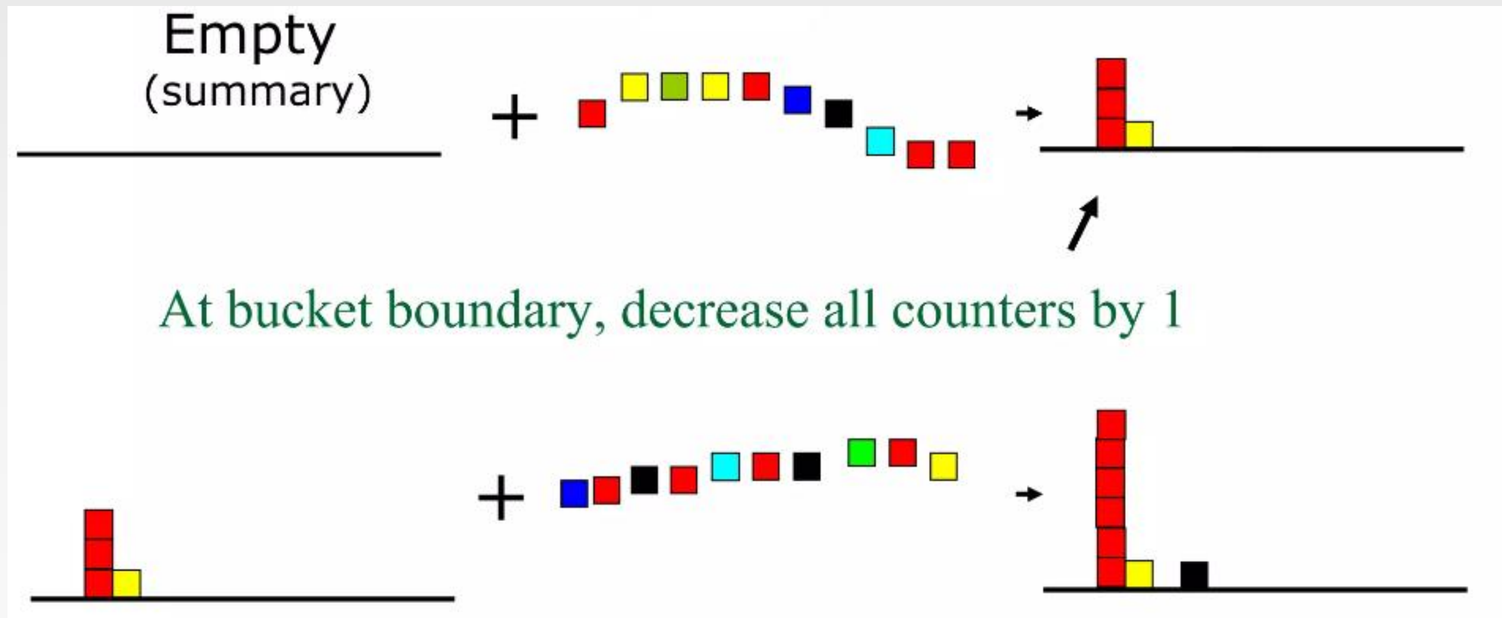



- Step 3: Repeat – Update counters and after each window, decrement all counters by 1



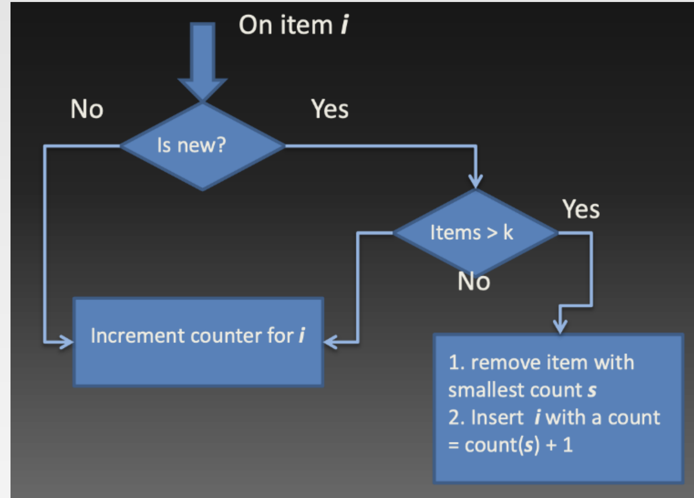
# Lossy Counting



# The Space-Saving Algorithm

- Keep k = 1/ε item names and counts, initially zero
- On seeing new item:
  - If it has a counter, increment counter
  - If not, replace item with least count, increment count

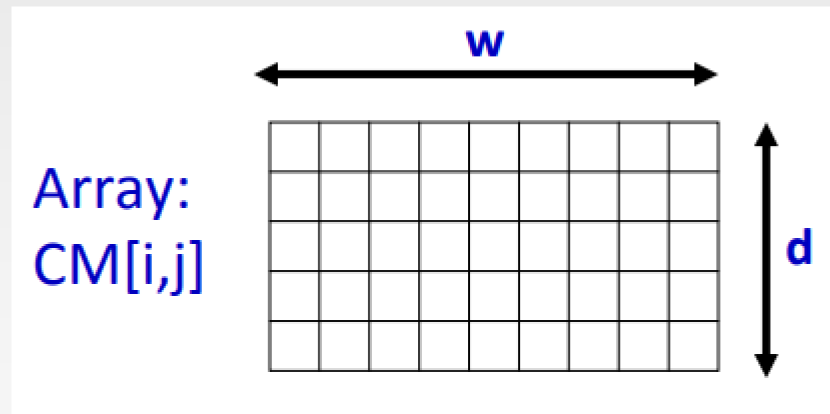



http://romania.amazon.com/techon/presentations/DataStreamsAlgorithms_FlorinManolache.pdf

- Analysis:
  - Smallest counter value, min, is at most εn
  - True count of an uncounted item is between 0 and min
  - Any item x whose true count > εn is stored
- So: Find all items with count > εn, error in counts ≤ εn

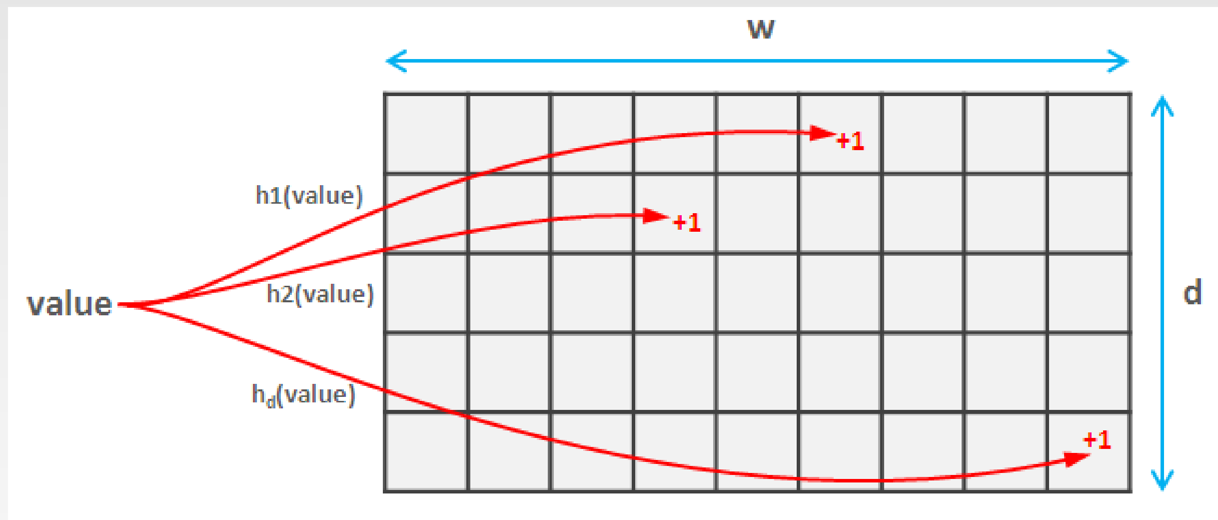
# Count-Min Sketch

- In general, model input stream as a vector x of dimension U
  - x[i] is frequency of element I
- The count-min sketch has two parameters, the number of buckets w and the number of hash functions d
- Creates a small summary as an array of w × d in size
- Use d hash function to map vector entries to [1..w]



# Count-Min Sketch

- The count-min-sketch supports two operations: Inc(x) and Count(x)
- The operation Count(x) is supposed to return the frequency count of x, meaning the number of times that Inc(x) has been invoked in the past
- The code for Inc(x) is simply:
  - for i = 1, 2, . . . , d: increment CMS[i][hi(x)]




- The code for Count(x) is simply:
  - return $min_{i=1}^{d}$CMS[i][hi(x)]

https://www.geeksforgeeks.org/count-min-sketch-in-java-with-examples/

# Sample Questions

- Assume that we have 5 buckets and three hash functions:
  - h0(str) = str.length * 2 mod 5
  - h1(str) = str.length mod 5
  - h2(str) = (str[0]-'a') mod 5

Given you a stream of terms: "big", "data", "data", "set", "data", "analytics", show the steps of building the CM-Sketch. Then, use the built CM-sketch to get the count for word "data".

- Solution:
  - big: h0 = 1, h1 = 3, h2 = 1
  - data: h0 = 3, h1 = 4, h2 = 3
  - set: h0 = 1, h1 = 3, h2 = 3
  - analytics: h0 = 3, h1 = 4, h2 = 0

# Solution

Initially:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 0 | 0 | 0 | 0 |
| h1 | 0 | 0 | 0 | 0 | 0 |
| h2 | 0 | 0 | 0 | 0 | 0 |

big:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 1 | 0 | 0 | 0 |
| h1 | 0 | 0 | 0 | 1 | 0 |
| h2 | 0 | 1 | 0 | 0 | 0 |

data:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 1 | 0 | 1 | 0 |
| h1 | 0 | 0 | 0 | 1 | 1 |
| h2 | 0 | 1 | 0 | 1 | 0 |

data:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 1 | 0 | 2 | 0 |
| h1 | 0 | 0 | 0 | 1 | 2 |
| h2 | 0 | 1 | 0 | 2 | 0 |

set:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 2 | 0 | 2 | 0 |
| h1 | 0 | 0 | 0 | 2 | 2 |
| h2 | 0 | 1 | 0 | 3 | 0 |

data:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 2 | 0 | 3 | 0 |
| h1 | 0 | 0 | 0 | 2 | 3 |
| h2 | 0 | 1 | 0 | 4 | 0 |

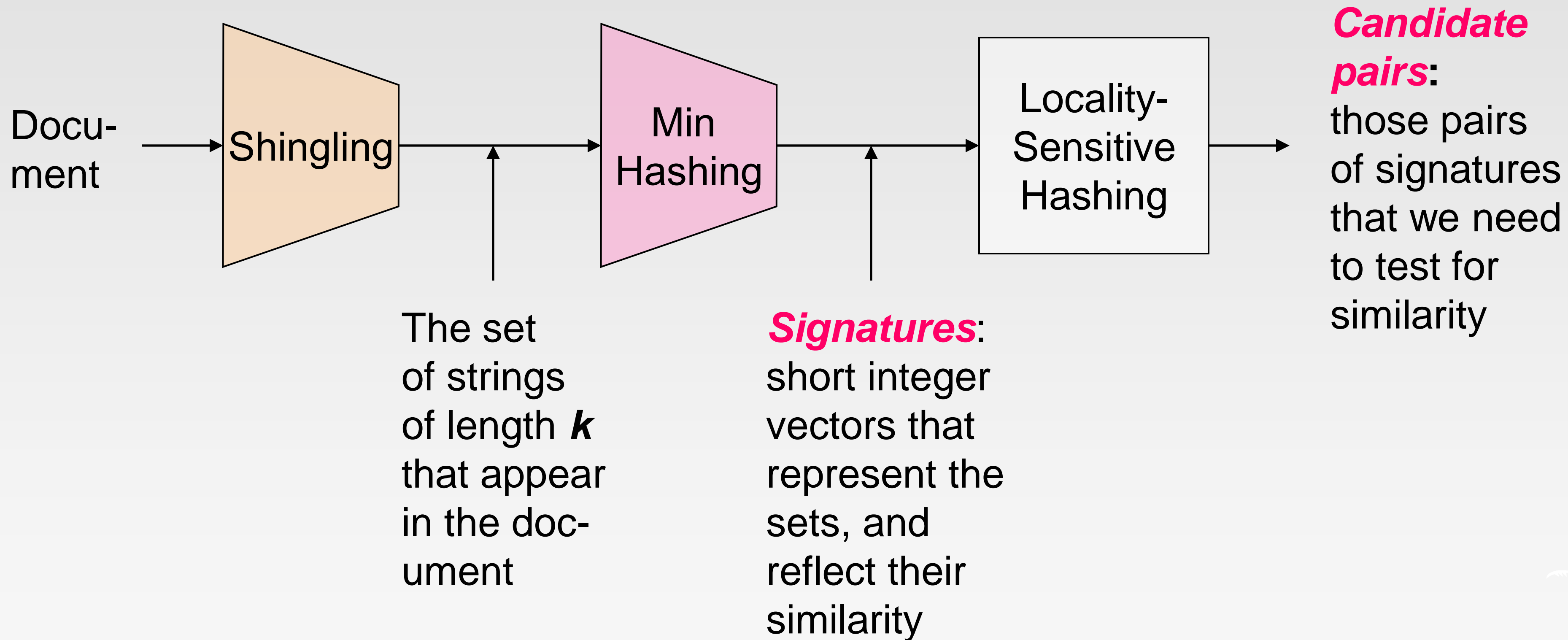
analytics:

| | b0 | b1 | b2 | b3 | b4 |
|---|---|---|---|---|---|
| h0 | 0 | 2 | 0 | 4 | 0 |
| h1 | 0 | 0 | 0 | 2 | 4 |
| h2 | 1 | 1 | 0 | 4 | 0 |

Min(CMS[0][3], CMS[1][4], CMS[2][3])=4, which is not the correct count.

# Finding Similar Items

❖ The Big Picture



# Shingling

- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
  - Tokens can be characters, words or something else, depending on the application
  - Assume tokens = characters for examples

- **Example: k=2**; document $\mathbf{D_1}$ = abcab
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {ab, bc, ca}

- Documents that are intuitively similar will have many shingles in common.
  - Example: k=3, “The dog which chased the cat” versus “The dog that chased the cat”.
    - Only 3-shingles replaced are g_w, _wh, whi, hic, ich, ch_, and h_c.

# Min-Hash Signatures

- **Pick K=100 random permutations of the rows**
- Think of ***sig*(C)** as a column vector
- **s*ig*(C)[i] =** according to the *i*-th permutation, the index of the first row that has a 1 in column *C*

$$sig(C)[i] = \min(\pi_i(C))$$

- **Note:** The sketch (signature) of document *C* is small **~100 bytes!**
- **We achieved our goal! We "compressed" long bit vectors into short signatures**

# Implementation Example

| Row | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $x+1 \mod 5$ | $3x+1 \mod 5$ |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 2 | 4 |
| 2 | 0 | 1 | 0 | 1 | 3 | 2 |
| 3 | 1 | 0 | 1 | 1 | 4 | 0 |
| 4 | 0 | 0 | 1 | 0 | 0 | 3 |

- 0. Initialize all ***sig(C)[i]* = ∞**

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $h_2$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

- Row 0: we see that the values of $h_1(0)$ and $h_2(0)$ are both 1, thus $sig(S_1)[0] = 1$, $sig(S_1)[1] = 1$, $sig(S_4)[0] = 1$, $sig(S_4)[1] = 1$,

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | $\infty$ | $\infty$ | 1 |
| $h_2$ | 1 | $\infty$ | $\infty$ | 1 |

- Row 1, we see $h_1(1) = 2$ and $h_2(1) = 4$, thus $sig(S_3)[0] = 2$, $sig(S_3)[1] = 4$

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | $\infty$ | 2 | 1 |
| $h_2$ | 1 | $\infty$ | 4 | 1 |

# Implementation Example

| Row | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $x+1 \mod 5$ | $3x+1 \mod 5$ |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 2 | 4 |
| 2 | 0 | 1 | 0 | 1 | 3 | 2 |
| 3 | 1 | 0 | 1 | 1 | 4 | 0 |
| 4 | 0 | 0 | 1 | 0 | 0 | 3 |

- Row 2: $h_1(2) = 3$ and $h_2(2) = 2$, thus sig($S_2$)[0] = 3, sig($S_2$)[1] = 2, no update for $S_4$

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 2 | 1 |
| $h_2$ | 1 | 2 | 4 | 1 |

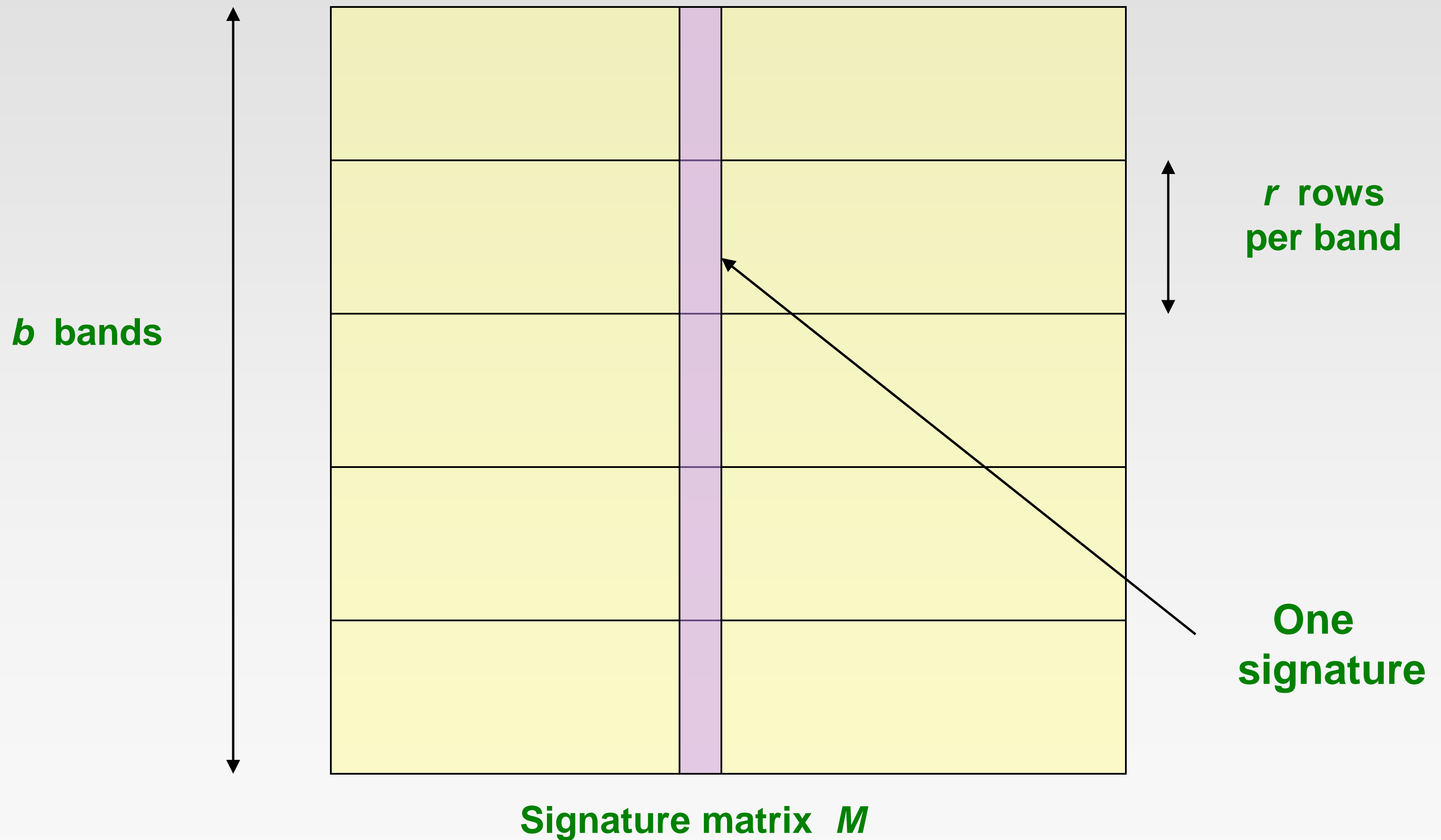
- Row 3: $h_1(3) = 4$ and $h_2(3) = 0$, update sig($S_1$)[1] = 0, sig($S_3$)[1] = 0, sig($S_4$)[1] = 0,

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 2 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

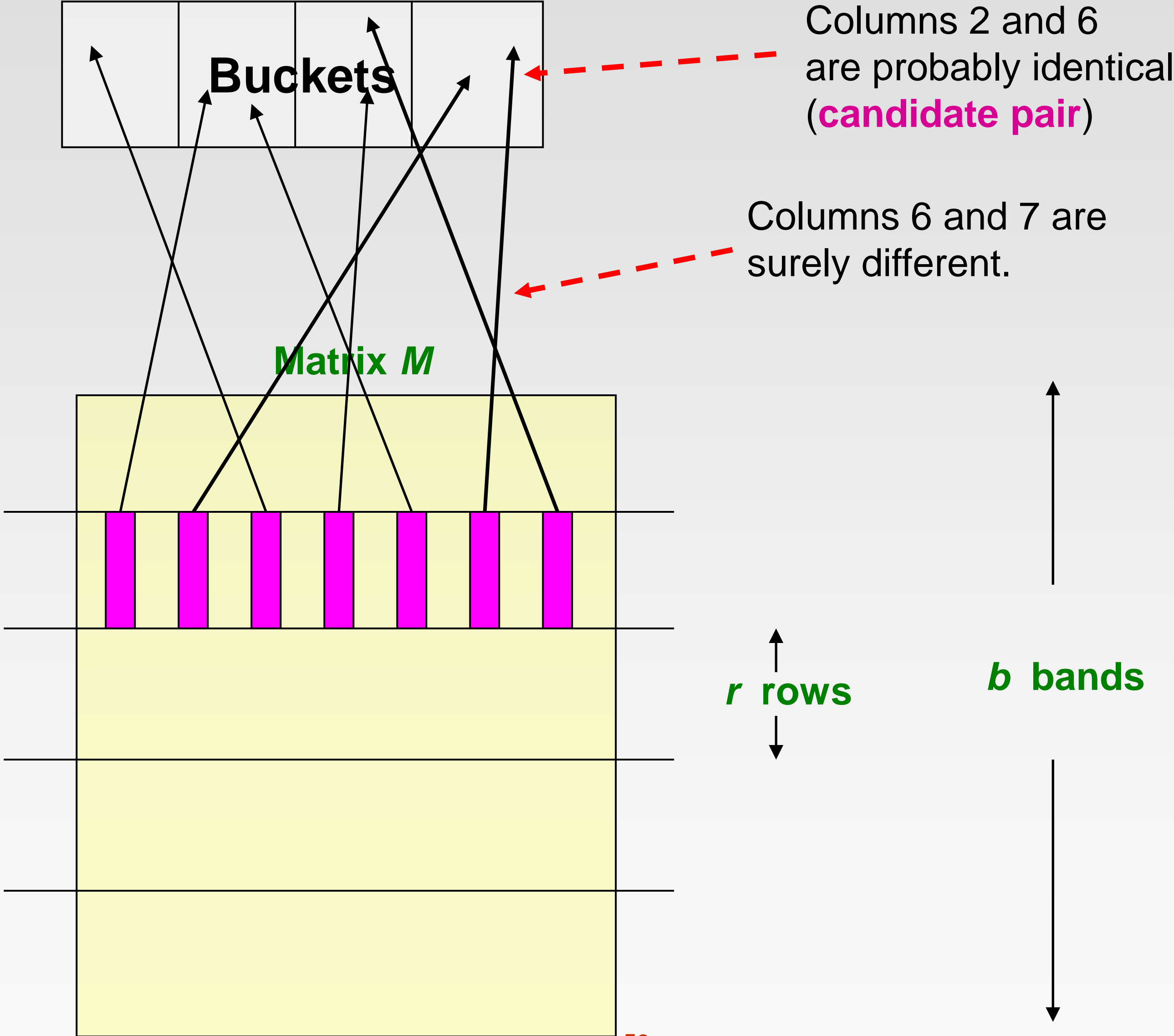
- Row 4: $h_1(4) = 0$ and $h_2(4) = 3$, update sig($S_3$)[0] = 0,

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 0 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

# Partition *M* into *b* Bands



# Hashing Bands



# *b* bands, *r* rows/band

- The probability that the minhash signatures for the documents agree in any one particular row of the signature matrix is $t$ ($sim(C_1, C_2)$)

- Pick any band ($r$ rows)
  - Prob. that all rows in band equal = $t^r$
  - Prob. that some row in band unequal = $1 - t^r$

- Prob. that no band identical = $(1 - t^r)^b$

- Prob. that at least 1 band identical = $1 - (1 - t^r)^b$

# Sample Questions

- k-Shingles:

Consider two documents A and B. Each document's number of token is O(n). What is the runtime complexity of computing A and B's k-shingle resemblance (using Jaccard similarity)? Assume that comparison of two k-shingles to assess their equivalence is O(k). Express your answer in terms of n and k.

Answer:

Assuming $n >> k$,

Time to create shingles = $O(n)$

Time to find intersection (using brute force algorithm) = $O(kn^2)$

Time to find union = $O(1)$ // computed as: $n + n - |intersection|$

Total time = $(kn^2)$

# Sample Questions

- MinHash:

We want to compute min-hash signature for two columns, $C_1$ and $C_2$ using two pseudo-random permutations of columns using the following function:

$$h_1(n) = 3n + 2 \bmod 7$$

$$h_2(n) = 2n - 1 \bmod 7$$

| Row | $C_1$ | $C_2$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |
| 2 | 0 | 1 |
| 3 | 0 | 0 |
| 4 | 1 | 1 |
| 5 | 1 | 1 |
| 6 | 1 | 0 |

Here, n is the row number in original ordering. Instead of explicitly reordering the columns for each hash function, we use the implementation discussed in class, in which we read each data in a column once in a sequential order, and update the min hash signatures as we pass through them.

Complete the steps of the algorithm and give the resulting signatures for $C_1$ and $C_2$.

# Solution

| Row | $C_1$ | $C_2$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |
| 2 | 0 | 1 |
| 3 | 0 | 0 |
| 4 | 1 | 1 |
| 5 | 1 | 1 |
| 6 | 1 | 0 |

$h_1(n) = 3n + 2 \mod 7$
$h_2(n) = 2n - 1 \mod 7$

| | Sig1 | Sig2 |
|---|---|---|
| | ∞ | ∞ |
| | ∞ | ∞ |
| $h1(0) = 2$ | ∞ | 2 |
| $h2(0) = 6$ | ∞ | 6 |
| $h1(1) = 5$ | 5 | 2 |
| $h2(1) = 1$ | 1 | 6 |
| $h1(2) = 1$ | 5 | 1 |
| $h2(2) = 3$ | 1 | 3 |
| $h1(4) = 0$ | 0 | 0 |
| $h2(4) = 0$ | 0 | 0 |

# Sample Questions

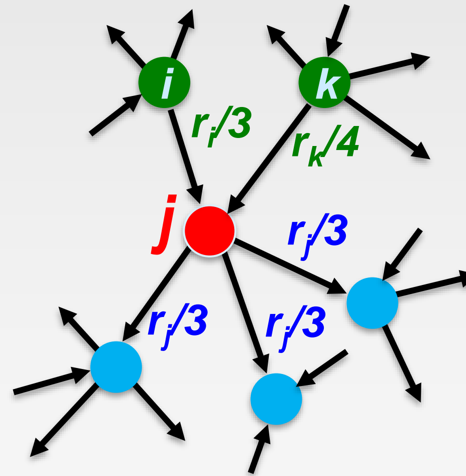
- Suppose we wish to find similar sets, and we do so by minhashing the sets 10 times and then applying locality-sensitive hashing using 5 bands of 2 rows (minhash values) each. If two sets had Jaccard similarity 0.6, what is the probability that they will be identified in the locality-sensitive hashing as candidates (i.e. they hash at least once to the same bucket)? You may assume that there are no coincidences, where two unequal values hash to the same bucket. A correct expression is sufficient: you need not give the actual number.

- Solution: **$1 - (1 - t^r)^b$**
  - $1 - (1 – 0.6^2)^5$

# Simple Recursive Formulation

- Each link's vote is proportional to the **importance** of its source page
- If page ***j*** with importance $r_j$ has ***n*** out-links, each link gets $r_j / n$ votes
- Page ***j***'s own importance is the sum of the votes on its in-links

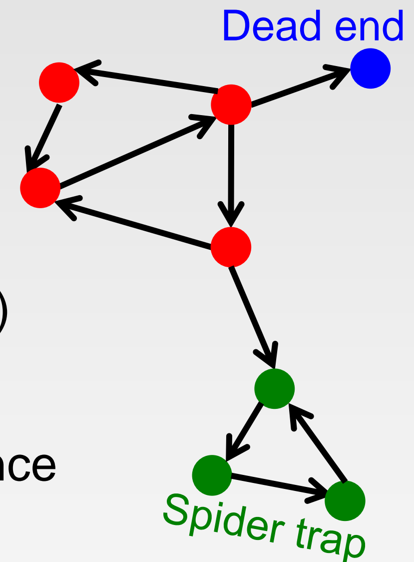
$$r_j = r_i/3 + r_k/4$$



# PageRank: Problems

2 problems:

- **(1)** Some pages are **dead ends** (have no out-links)
  - Random walk has "nowhere" to go to
  - Such pages cause importance to "leak out"




- **(2) Spider traps:** (all out-links are within the group)
  - Random walked gets "stuck" in a trap
  - And eventually spider traps absorb all importance

# PageRank: The Complete Algorithm

- **Input:** Graph $G$ and parameter $\beta$
  - Directed graph $G$ (can have **spider traps** and **dead ends**)
  - Parameter $\beta$
- **Output:** PageRank vector $r^{new}$
  - **Set:** $r_j^{old} = \frac{1}{N}$
  - **repeat until convergence:** $\sum_j \left| r_j^{new} - r_j^{old} \right| > \varepsilon$
    - $\forall j$: $r'^{new}_j = \sum_{i \to j} \beta \frac{r_i^{old}}{d_i}$

      $r'^{new}_j = 0$ if in-degree of $j$ is $0$
    - **Now re-insert the leaked PageRank:**

      $\forall j$: $r_j^{new} = r'^{new}_j + \frac{1-S}{N}$ **where:** $S = \sum_j r'^{new}_j$
    - $r^{old} = r^{new}$

If the graph has no dead-ends then the amount of leaked PageRank is **1-β**. But since we have dead-ends the amount of leaked PageRank may be larger. We have to explicitly account for it by computing **S**.

# Sparse Matrix Encoding

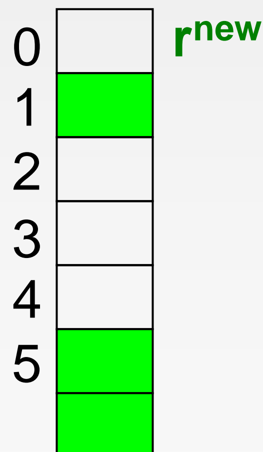
- Encode sparse matrix using only nonzero entries
  - Space proportional roughly to number of links
  - Say 10N, or 4*10*1 billion = 40GB
  - **Still won't fit in memory, but will fit on disk**

| source node | degree | destination nodes |
|---|---|---|
| 0 | 3 | 1, 5, 7 |
| 1 | 5 | 17, 64, 113, 117, 245 |
| 2 | 2 | 13, 23 |

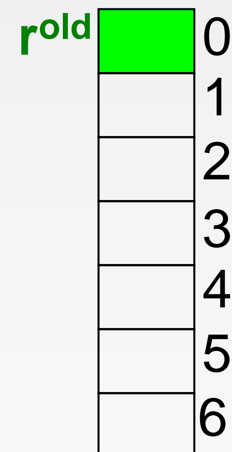
# Basic Algorithm: Update Step

- Assume enough RAM to fit $r^{new}$ into memory
  - Store $r^{old}$ and matrix **M** on disk
- 1 step of power-iteration is:

**Initialize** all entries of $\mathbf{r^{new}} = (1-\beta) / N$
For each page $i$ (of out-degree $d_i$):
  Read into memory: $i$, $d_i$, $dest_1$, …, $dest_{d_i}$, $r^{old}(i)$
  For $j = 1 \ldots d_i$
    $r^{new}(dest_j) \mathrel{+}= \beta \, r^{old}(i) / d_i$



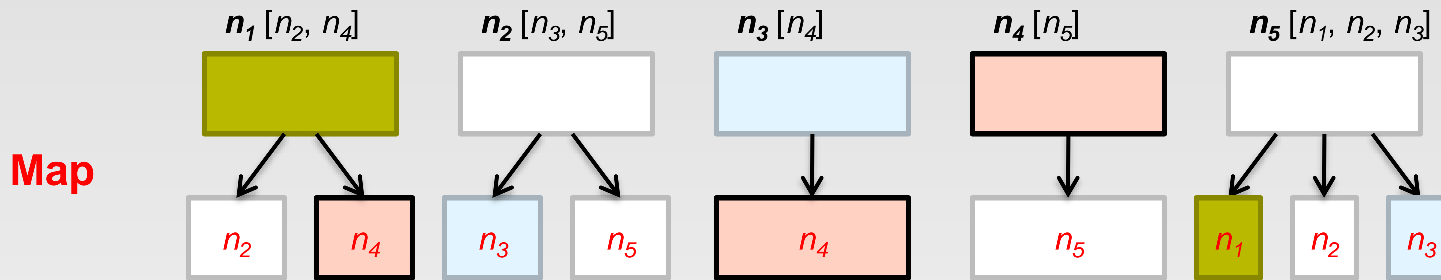

| source | degree | destination |
|---|---|---|
| 0 | 3 | 1, 5, 6 |
| 1 | 4 | 17, 64, 113, 117 |
| 2 | 2 | 13, 23 |

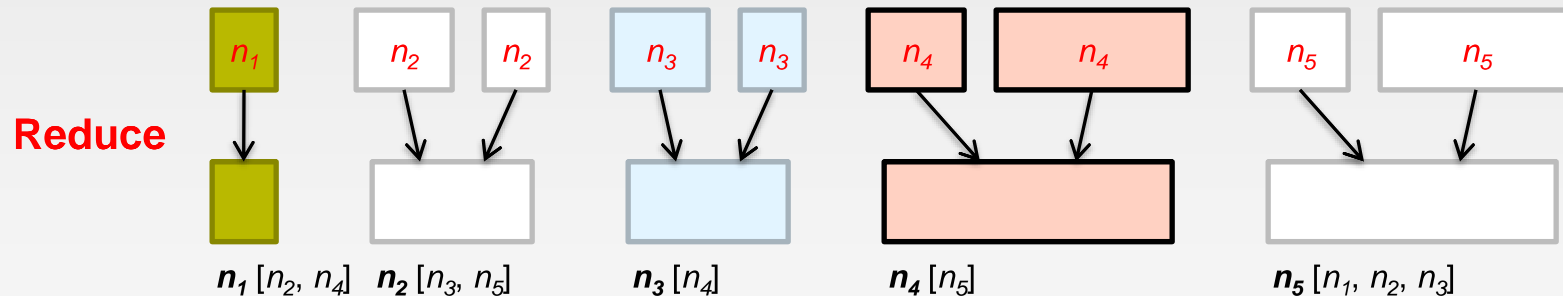


# PageRank Pseudo-Code

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sums incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

# PageRank in MapReduce (One Iteration)



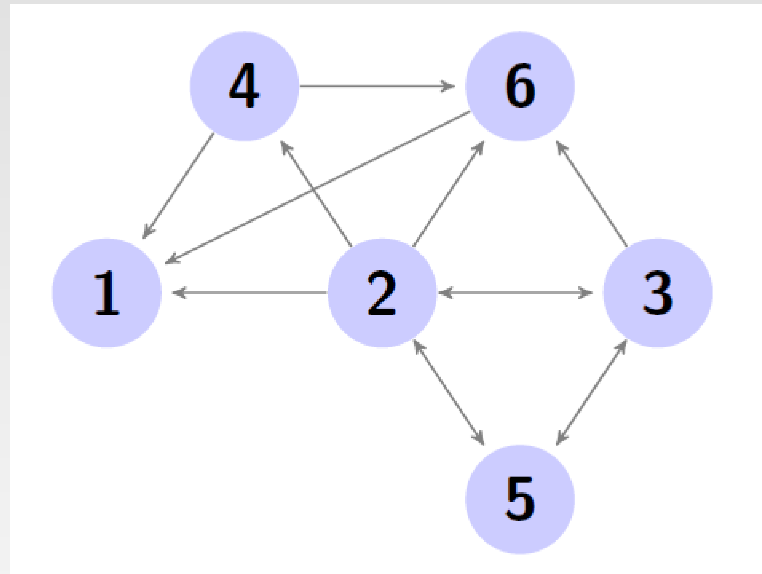



# Complete PageRank

- Two additional complexities
  - What is the proper treatment of dangling nodes?
  - How do we factor in the random jump factor?
- Solution:
  - If a node's adjacency list is empty, distribute its value to all nodes evenly.
    - In mapper, for such a node *i*, emit (nid m, $r_i$/N) for each node m in the graph
  - Add the teleport value
    - In reducer, M.PageRank = $\beta * s + (1- \beta) / N$

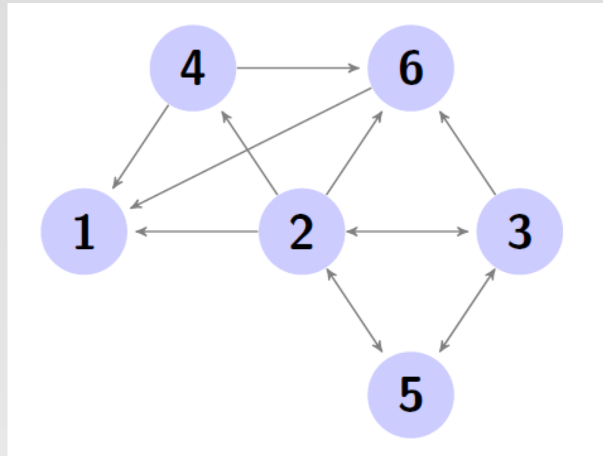
# Sample Questions

- A directed graph G has the set of nodes {1,2,3,4,5,6} with the edges arranged as follows.




- Set up the PageRank equations, assuming β = 0.8 (jump probability = 1- β). Denote the PageRank of node a by r(a).

# Solution




$$r(1) = 0.8(\frac{1}{6} \cdot r(1) + \frac{1}{2} \cdot r(4) + r(6) + \frac{1}{5} \cdot r(2)) + \frac{0.2}{6} \quad (1)$$

$$r(2) = 0.8(\frac{1}{6} \cdot r(1) + \frac{1}{3} \cdot r(3) + \frac{1}{2} \cdot r(5)) + \frac{0.2}{6} \quad (2)$$

$$r(3) = 0.8(\frac{1}{6} \cdot r(1) + \frac{1}{5} \cdot r(2) + \frac{1}{2} \cdot r(5)) + \frac{0.2}{6} \quad (3)$$

$$r(4) = 0.8(\frac{1}{6} \cdot r(1) + \frac{1}{5} \cdot r(2)) + \frac{0.2}{6} \quad (4)$$

$$r(5) = 0.8(\frac{1}{6} \cdot r(1) + \frac{1}{5} \cdot r(2) + \frac{1}{3} \cdot r(3)) + \frac{0.2}{6} \quad (5)$$

$$r(6) = 0.8(\frac{1}{6} \cdot r(1) + \frac{1}{5} \cdot r(2) + \frac{1}{3} \cdot r(3) + \frac{1}{2} \cdot r(4)) + \frac{0.2}{6} \quad (6)$$

# Thank you!